Algebraic Semantics of the C Preprocessor and Correctness of its Refactorings

Alejandra Garrido, José Meseguer, and Ralph Johnson

Technical Report UIUCDCS-R-2006-2688
(Engineering No. UILU-ENG-2006-1717)
Department of Computer Science.
University of Illinois at Urbana-Champaign, Urbana IL 61801, USA.
{garrido,meseguer,johnson}@cs.uiuc.edu
February 2006

Abstract. Refactoring has become a popular technique for the development and maintenance of object-oriented systems. We have been working on the refactoring of C programs, including the C preprocessor (Cpp), and we have built CRefactory, a refactoring tool for C programs. The independence of Cpp from the underlying programming language complicates the analysis and refactoring of programs that use Cpp. Nevertheless, that independence is helpful when implementing refactorings on Cpp directives.
While refactorings are defined as "behavior preserving transformations", there is usually no formal proof of their correctness. By using rewriting logic and its Maude implementation, we have formally specified the semantics of Cpp and also of some refactorings on Cpp directives. The specifications are then used to formally prove refactoring correctness. This paper describes the formal specifications of Cpp and of three of its refactorings, and presents the proofs of their correctness.

1 Introduction

Refactoring allows improving code design, making it more readable, reusable and flexible to subsequent semantic changes [1]. The concept of refactoring has evolved from that of "software restructuring" [2], into cataloged syntactic transformations, applied in small steps, usually with the assistance of an interactive tool. Moreover, refactorings are assumed to preserve the behavior of the program. A refactoring preserves program behavior when the versions of the program before and after refactoring are semantically equivalent, that is, the mapping of input to output values remains the same [3]. Typical examples of refactorings are "Rename Variable" or "Extract Method" [1].

While working on a refactoring tool for C programs, we encountered the challenge of having to deal with the C preprocessor (Cpp). Cpp is heavily used in C programs since it enhances C with several facilities, like the definition of constants, the abbreviation of repetitive or complicated constructs, the manipulation of types as first class objects, program configuration with conditional

compilation, and partition of programs in multiple files, among plenty of other uses. However, Cpp's ability to perform unstructured source code manipulations complicates the understanding of C programs by programmers and tools [4].

The main difficulty with Cpp arises from its independence of the C language or any other programming language that uses it (like Fortran or C++). This independence, however, means that Cpp is generic, and can be used as a preprocessor language for any programming language. Therefore, programs that use Cpp have a mix of two languages. Before a program can be parsed and compiled, it needs to be preprocessed, so that Cpp directives are evaluated and removed. However, if a refactoring tool preprocesses and parses a program and transformations are applied to the result, the Cpp directives that were originally in the program may be irrevocably lost. Without the features provided by Cpp, the preprocessed version of the program may easily become unmaintainable.

We have built a refactoring tool for C programs, called CRefactory, that preserves Cpp directives in a program and its refactorings, by integrating them with C language constructs in the program representations [5]. Moreover, CRefactory allows applying refactorings on Cpp directives themselves. Examples of these refactorings are: "Extract File", "Inline File", "Rename Macro", "Extract Macro", "Remove Condition", etc. [5]. These refactorings are, in general, simpler than the refactorings on pure C, because the syntax of Cpp is relatively simple, transformations do not require a parse tree, and proving their correctness amounts to proving that the output of Cpp, a sequence of tokens without Cpp directives, remains the same. Indeed, our Cpp formal semantics and proof of correctness for refactorings can be exploited in a generic way for any language using Cpp (e.g., C, C++, Fortran).

With those goals in mind, what is necessary is to choose a suitable formal language to specify Cpp's syntax and semantics and to specify refactorings. Mathematical proofs of refactoring correctness can then be based on such semantics. We have found Maude [6] specially suitable for this task. Maude provides a framework for rewriting logic semantics and an environment where specifications can be directly executed. In particular, Maude's support for membership equational logic [7], which allows defining partial functions and data types whose data satisfy sophisticated semantic conditions, has been very useful in defining the Cpp semantics. Our Cpp equational specification is indeed executable, yielding a Cpp interpreter.

Related Work. Eelco Visser discusses the application of term rewriting to the wider area of program transformation and surveys different approaches and extensions [8]. In particular, his paper covers different strategies for tree parsing, tree traversal and programmable transformations, like those in Stratego [9] and ASF+SDF [10]. The focus is on optimizations-like transformations and on functional languages.

Paulo Borba et. al. propose basic algebraic programming laws for a language similar to a subset of sequential Java [11]. The laws are proven to be sound and complete, and are composed to create refactorings like Pull Up/Push Down

Method. The specification of the syntax of the language and the laws is based on Dijkstra's language of guarded commands.

Ralf Sasse has used Maude and the JavaFAN formal specification of Java written in Maude [12] to cross-validate the rules of a programming language proof calculus called *KeY* [13, 14]. Many of the *KeY* rules considered in [13, 14] are indeed refactoring-like program transformation rules.

This paper is organized as follows. Section 2 provides a brief background on Cpp and Maude. Section 3 outlines the Maude specification of Cpp, which is listed entirely in the Appendix. Section 4 presents the formal specification of three refactorings: 'Inline File', 'Extract File' and 'Rename Macro'. These refactorings are then proved correct in Sect. 5. The last section outlines conclusions and future work.

2 Preliminaries on Cpp and Rewriting Semantics

2.1 The C Preprocessor (Cpp)

Cpp is controlled by special commands called *preprocessor directives*. Preprocessor directives start with '`#`' and their syntax is completely independent of the syntax of the C language [15]. The most important and used directives in Cpp are: `#include`, for the inclusion of header files, `#define`, to create macros with or without parameters, and `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` and `#endif`, to control the inclusion of code based on configuration setting [15].

Preprocessing occurs as a separate phase before compilation. Cpp receives as input a source file, directories for header files (that get included by the source file) and possibly some macro definitions, and transforms the input by a series of textual replacements. These replacements include removing comments, converting the input file into a sequence of tokens (tokenization), executing directives and expanding macros [16]. The output of Cpp is a sequence of tokens corresponding to the underlying programming language, without directives.

Figure 1 shows an example of a program composed of two files (a source file "`main.c`" and a header file "`defs.h`") and the result of preprocessing it (on the right), assuming a configuration where '`_UX`' has been defined as a macro, so the condition of the `#ifdef` directive is true and the code inside its branch is included in the output. The `#include` directive causes the contents of the file "`defs.h`" to be copied at that position. The definition for macro `ST` is used when called from the `case` statements. To expand the macro `ST`, the operator `##` directs Cpp to concatenate the name received as argument with the string '`Status`'.

2.2 Rewriting Semantics of Programming Languages

Rewriting logic provides a powerful framework for specifying the semantics of both sequential and concurrent programming languages by unifying SOS and equational semantics [17]. Moreover, the Maude environment allows the direct

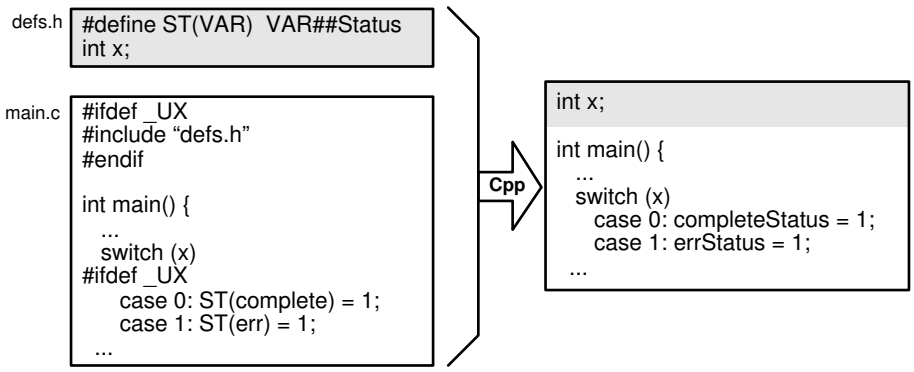

Fig. 1. Example of the use of Cpp directives

execution of semantic specifications as interpreters. Once the semantics of a language has been defined, it is then possible to develop special-purpose analysis tools based on that semantics [17].

In our case, we have defined in Maude both, the semantics of Cpp and the refactoring rules. Both are executable, so we are able to evaluate programs and perform refactoring. Moreover, mathematical proofs of the correctness of the refactorings can be based on the formal Cpp semantics. Since there is no concurrency involved in Cpp, its specification can be entirely defined with equations. More precisely, the semantics of Cpp is defined as a membership equational theory $(\Sigma_{Cpp}, (E \cup A)_{Cpp})$, where the signature Σ_{Cpp} specifies Cpp's syntax and auxiliary operators, and the equations and memberships in $(E \cup A)_{Cpp}$ specify the operational semantics of Cpp, with E the set of (possibly conditional) equations and memberships, and A a set of axioms for associativity, commutativity and identity of some operators in Σ_{Cpp}.

The rewriting logic semantics of a programming language is defined in Maude [6] by a sequence of modules [17, 18]. There are specific modules for the syntax of the language, other modules that specify the semantics of each feature, and a third sequence of modules for auxiliary data structures and operations. The semantics of Cpp is defined on a data structure that we call CppState, which stores state information necessary to define each language construct.

There are different styles to specifying a language, as described in [17]. We have chosen to merge evaluation and state update in a single operation called state. The complete Maude specification of Cpp is listed in the Appendix, and described in detail in [5]. The next section describes the main operations and sorts, and some semantic equations.

3 The C Preprocessor

In order to correctly specify refactorings, it is first necessary to shed some light into the somewhat obscure manipulations of Cpp. We do so by formally specifying the semantics of Cpp as an equational theory in Maude. As far as we know, this is the only formal specification ever written for Cpp.

Input to Cpp. The input to Cpp is modelled through a set of files that compose the program, with sort CFilesMap. That is, instead of directories for header files, the CFilesMap has entries for each header file included, besides the source file, and instead of giving macros in the command line, they are defined in a header file. Each element of the CFilesMap is a CFile, of the form:

```
op [_:_] : String LineSeq -> CFile .
```

that is, a CFile is a pair with the file name as first component and its contents, a sequence of lines (with sort LineSeq), as second component. For example, the program shown in Fig. 1 would be represented as a CFilesMap with two elements (using empty juxtaposition associative-commutative syntax to form their union): `[‘‘main.c’’ : #if ...] [‘‘defs.h’’ : #define ...]`.

A Line is the unit of processing for Cpp, where a Line is either a Cpp directive or a sequence of tokens (TokenSequence) followed by a `cr` (carriage return). The tokens in a TokenSequence correspond to the underlying language and may include macro calls. The operation nilLS represents the empty line sequence and nil is the empty token sequence. The empty CFilesMap is represented by empty.

The preprocess operation. The external interface of Cpp is modelled with the operation preprocess. The syntax and typing of this operation are:

```
op preprocess : CFilesMap String -> TokenSequence .
```

that is, it receives a CFilesMap and a String as arguments, and outputs a TokenSequence in which directives and macros have been stripped out. The second argument represents the name of the source file, among the ones in the CFilesMap, from where to start the preprocessing.

The semantics of the preprocess operation is to first create the initial CppState and then call the state operation on the LineSeq of the starting file and the initial CppState.

The CppState. The state is defined as a multiset of state attributes, that is, as an associative and commutative data structure with the empty mutiset as the identity element. The attributes have sort CppStateAtribute, and are specified with constructors that take as argument the value that each one stores. The name of constructor and type of argument for each attribute are:

– files(CFilesMap). The set of files that compose the program. It is used to execute `#include` directives.

- macroTbl(MacroTable). The set of macro definitions that are currently active. This table is populated with `#define` directives and is used to check for and expand macros.
- curMacroCalls(IdentifierListP). The list of macros currently being expanded.
- skip(Bool). Used to determine if the current line is in the true branch of a Cpp conditional or should be skipped (i.e., stripped out from the output).
- nestLevelOfSkipped(Nat). Used to count the depth of nesting of conditional directives in a false branch (a branch being skipped).
- branchTaken(Bool). Used to determine if the true branch of the current Cpp conditional construct has already been taken.
- outputStream(TokenSequence). This is the output of Cpp that gets incrementally constructed during preprocessing and is returned at the end.

The state operation. This operation has two variants, one receiving a Line and the other receiving a LineSeq as first argument. The syntax and basic equations defining its semantics are:

```
op state : Line CppState -> CppState .
op state : LineSeq CppState -> CppState .
eq state(nilLS, S) = S .
eq state(L LS, S) = state(LS, state(L, S)) .
```

That is, when the state operation is called on an empty LineSeq (nilLS), it returns the current CppState without changes. When it is called on a non-empty LineSeq, it calls itself recursively on the first line of the LineSeq, and the result is passed to a new state operation on the rest of the lines.

Cpp directives. A Cpp directive is a line that starts with '`#`' followed by the directive name. There is a sort CppDirective and subsorts for each specific directive. Moreover, CppDirective is defined as a subsort of Line.

The following subsections describe the syntax and semantics of the Cpp directives: `#include`, `#define`, and of the set of conditional directives.

File inclusion. The `#include` directive allows merging a program conveniently divided into separate files, obtaining a single 'compilation unit'. The name of the file to be included is denoted after the `#include` keyword with one of three possible forms: "$filename$" (which usually denotes a header file in the same package or subsystem), $< filename >$ (to refer to library or standard implementation files) or a macro call that expands to one of the previous forms (the latter are called "computed includes" [16]). The specification of the syntax of the `#include` directive appears in module `INCLUDE-SYNTAX` (found in the Appendix).

This directive causes Cpp to scan the specified file as input before continuing with the rest of the current file. The equation defining its semantics, for the case of a String argument and when the line is not being skipped, is:

```
eq state(#include FN cr, (files(FS), skip(false), S))
  = state(readFile(FN, FS), (files(FS), skip(false), S)) .
```

The operation readFile used to execute the `#include` directive returns the LineSeq associated with the file `FN` in the CFilesMap `FS`, or nilLS if `FN` is not in `FS`.

To facilitate proofs of refactoring correctness, we impose additional requirements on the sort CFilesMap. Specifically: (i) there cannot be duplicated file names; and (ii) there cannot be a cycle of file inclusion dependencies (a file cannot include itself directly or indirectly). These properties are specified with a conditional membership axiom [6] for the sort CFilesMap:

```
cmb ([N : LS] FM) : CFilesMap  if FM : CFilesMap /\
   (not N in dom(FM)) /\
   (not N in (includedFiles(LS) reach(FM, includedFiles(LS)))).
```

That is, `([N : LS] FM)` is a CFilesMap if `FM` is a CFilesMap, and `N` does not belong to the domain of `FM`, and `N` is not in the set of files included directly in the line sequence `LS`, or indirectly by the files 'reachable' from `LS`.

Macro definition. The `#define` directive allows the definition of macros. A macro is a text fragment which has been given a name and can have arguments. The syntax of a macro is specified with:

```
op #define__cr : Identifier TokenSequence -> MacroDefDir.
op #define___cr : Identifier IdentifierListP TokenSequence
                  -> MacroDefDir.
```

The first argument represents the macro name and the TokenSequence is the replacement text. The second version is used for a macro with arguments, where IdentifierListP is a parenthesized comma-separated list of Identifiers. For example, the macro definition that appears in Figure 1 would be specified with:
`#define 'ST('VAR) ('VAR '## 'Status) cr`

This directive causes Cpp to substitute all occurrences of the macro name by its replacement text. For this purpose, Cpp executes a `#define` directive by adding an entry of sort MacroDef in the macro table. This is specified, for a macro without arguments, as follows:

```
op name_replText_ : Identifier TokenSequence -> MacroDef .
op [_:_] : Identifier MacroDef -> MacroTable .
op __ : MacroTable MacroTable -> MacroTable [assoc comm id: empty] .
eq state(#define I TS cr, (macroTbl(MT), skip(false), S))
  = macroTbl([I : (name I replText TS)] MT), skip(false), S .
```

When a token is found that matches the name of a macro in the macro table, the token is replaced by the associated TokenSequence, with appropriate argument substitution. If a macro currently being expanded appears again in the replacement text (as a direct or indirect self-reference), it is not macro-expanded again, which prevents infinite recursion. The CppState attribute curMacroCalls is used to prevent self-expansion.

There is also a `#undef` directive to 'undefine' or forget a macro definition. It takes the name of the macro as parameter and removes the entry for that name from the macro table.

Conditional Directives. These directives allow conditional code inclusion based on configuration settings. They instruct Cpp to evaluate a condition during preprocessing and, depending on the resulting truth value, select whether or not to include a piece of code in the output.

A conditional directive is one of the following: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` or `#endif`. The `#if`, `#ifdef` and `#ifndef` directives start a *Cpp conditional construct*, creating its first branch. The `#elif` and `#else` directives create additional branches on the Cpp conditional and the `#endif` ends the construct. The source text inside a branch may include other preprocessor directives. Consequently, Cpp conditionals can also be nested. Figure 2 shows an example taken from the file "`compiler.h`" in the Linux kernel (version 2.6.7).

```
#ifndef __ASSEMBLY__
#if __GNUC__ > 3
# include <linux/compiler-gcc+.h>
#elif __GNUC__ == 3
# include <linux/compiler-gcc3.h>
#elif __GNUC__ == 2
# include <linux/compiler-gcc2.h>
#else
# error Sorry, your compiler is too old/not recognized.
#endif
#endif
```

Fig. 2. Example of the use of conditional directives

The `#if` and `#elif` tokens are followed by a constant expression. Constant expressions are specified with the sort CondExp. There are a few modules that specify the available operations between CondExps. However, the argument of a `#if` or `#elif` in the specification is not a CondExp but a TokenSequence. The reason for this is that the condition can have macro calls, which expand to a TokenSequence that may not be a complete syntactical unit. It is easier to have everything expanded into a TokenSequence and then translate it to a CondExp to be evaluated.

The lines "`#ifdef` *id*" and "`#ifndef` *id*" are abbreviations of "`#if defined` *id*" and "`#if !(defined` *id*`)`" respectively. For example, the syntax of the `#ifdef` directive is specified by: `op #ifdef_cr : Identifier -> CondDir .`

The semantics of the state operation upon a `#ifdef` directive is to check if the Identifier is the name of a macro in the current macro table, and if it is, then continue in 'non-skipping mode' (`skip=false`) and mark that a branch in this conditional has been taken (`branchTaken=true`). If the Identifier has not been defined as a macro, the following code is skipped as if it were a comment, until the end of the branch is found. To prevent confusing the end of the branch with the end of a nested conditional, the state attribute nestLevelOfSkipped counts

the depth of nested conditionals. This behavior is specified with the following two equations:

```
var I : Identifier. var AMT : MacroTable. var S : CppState.
ceq state(#ifdef I cr, (macroTbl(AMT),skip(false),branchTaken(false),S))
    = macroTbl(AMT), skip(false), branchTaken(true), S
    if isMacro(I, AMT) .
ceq state(#ifdef I cr, (macroTbl(AMT),skip(false),nestLevelOfSkipped(0),
         branchTaken(false), S))
    = macroTbl(AMT), skip(true), nestLevelOfSkipped(1),
         branchTaken(false), S     if not isMacro(I, AMT) .
```

Similar equations exist for each one of the conditional directives, and can be found in the Appendix.

4 Cpp Refactorings and their Semantics

This section presents the rules of three refactorings on Cpp directives: 'Inline File', 'Extract File' and 'Rename Macro'. In the formal specification, there is a sort CppRefactoring and an operation <- that applies a CppRefactoring to a CFilesMap and returns a *transformed* CFilesMap, with the following syntax:

```
op _<-_ : CFilesMap CppRefactoring -> CFilesMap .
```

Each specific refactoring, InlineFileRefactoring, ExtractFileRefactoring and RenameMacroRefactoring, is defined as a subsort of CppRefactoring.

4.1 Inline File Refactoring

In a similar way as a function is inlined, this refactoring replaces the `#include` directive of a file by its contents, a LineSeq. Additionally, it removes the inlined file from the CFilesMap representing the program. The refactoring assumes that `R` is not computed through a macro call in the `#include` directive.

Figure 3 shows the equations that specify the transformation. The operation InlineFile receives the name of the file to be inlined as input (`R`), and if the file exists in the CFilesMap, it removes the file from it and calls removeIncludes on the rest of the CFilesMap. The operation removeIncludes calls remInclLines on the line sequence of each file, which in turn replaces the lines '`#include R cr`' by the line sequence of `R` (like executing the `#include` directive only on `R`). Note the use of the attribute `[owise]` (otherwise) in the last equation, which makes it applicable when all the previous have failed to apply, i.e., when the third argument is not `nilLS` or the first line is not a `#include` for `R`. The attribute `[owise]` can be desugared into an equivalent conditional specification [6].

4.2 Extract File Refactoring

In this refactoring, the user selects a piece of code to be moved into a new file. A `#include` directive for the new file is inserted at the place of the selection. Figure

```
fmod INLINE-FILE-REF is
  --- some module importation
  sort InlineFileRefactoring .
  subsort InlineFileRefactoring < CppRefactoring .
  op InlineFile : String -> InlineFileRefactoring .
  op removeIncludes : String LineSeq CFilesMap -> CFilesMap .
  op remInclLines : String LineSeq LineSeq -> LineSeq .
  vars R F : String . vars LSr LS : LineSeq . var L : Line . var FM : CFilesMap .
  eq ([R : LSr] FM) <- InlineFile(R) = removeIncludes(R, LSr, FM) .
  ceq FM <- InlineFile(R) = FM  if not containsFileName(R, FM) .
  eq removeIncludes(R, LSr, empty) = empty .
  eq removeIncludes(R, LSr, [F : LS] FM)
    = [F : remInclLines(R, LSr, LS)] removeIncludes(R, LSr, FM) .
  eq remInclLines(R, LSr, nilLS) = nilLS .
  eq remInclLines(R, LSr, (#include R cr) LS) = LSr ++ remInclLines(R, LSr, LS) .
  eq remInclLines(R, LSr, L LS) = L remInclLines(R, LSr, LS) [owise] .
endfm
```

Fig. 3. Module implementing Inline File refactoring

4 shows the module with the specification for this refactoring. The operation ExtractFile expects three parameters: the name of the file from where the line sequence is extracted (F:String), the line sequence to extract (LS2:LineSeq), and the name of the new file (NewF:String). The precondition for this refactoring is that NewF is not used as the name of another file in the CFilesMap FM. Moreover, ExtractFile is applied only if the arguments are valid, i.e., if F is the name of a file in the CFilesMap and if LS2 is part of the LineSeq of F and not empty. If the refactoring is applied, the CFile '[NewF:LS2]' is added to FM, and the operation extractAndAddInclude constructs a new line sequence for F, by replacing LS2 with a '#include NewF cr' line.

```
fmod EXTR-FILE-REF is
  --- some module importation
  sort ExtractFileRefactoring .
  subsort ExtractFileRefactoring < CppRefactoring .
  op ExtractFile : String LineSeq String -> ExtractFileRefactoring .
  op extractAndAddInclude : LineSeq LineSeq String -> LineSeq .
  op extractFrom : LineSeq LineSeq -> LineSeq .
  vars F NewF : String . vars LS1 LS2 : LineSeq .
  var FM : CFilesMap . vars L L' : Line .
  eq ([F : LS1] FM) <- ExtractFile(F, nilLS, NewF) = [F : LS1] FM .
  ceq ([F : LS1] FM) <- ExtractFile(F, LS2, NewF)
     = [F : extractAndAddInclude(LS1, LS2, NewF)] [NewF : LS2] FM
    if not (containsFileName(NewF, [F : LS1] FM)) and (LS2 inLS LS1) .
  eq ([F : LS1] FM) <- ExtractFile(F, LS2, NewF) = [F : LS1] FM [owise] .
  ceq FM <- ExtractFile(F, LS2, NewF) = FM  if not containsFileName(F, FM) .
  ceq extractAndAddInclude(L LS1, L' LS2, NewF)
     = L extractAndAddInclude(LS1, L' LS2, NewF) if L =/= L' .
  eq extractAndAddInclude(L LS1, L LS2, NewF)
     = #include NewF cr extractFrom(LS1, LS2) .
  eq extractFrom(LS1, nilLS) = LS1 .
  eq extractFrom(L LS1, L LS2) = extractFrom(LS1, LS2) .
endfm
```

Fig. 4. Module implementing Extract File refactoring

4.3 Rename Macro Refactoring

Renaming is probably the best known and used refactoring. The precondition is that the new name does not clash with any other symbol in the scope. Since in this paper we are only concerned with Cpp and how to prove correctness of Cpp's refactorings, we assume throughout that the old name of the macro is not used as the name of any entity of the underlying programming language (e.g., a C, C++ or Fortran variable). With this assumption, every occurrence of the old name is replaced by the new name. There may be more than one definition for the macro to be renamed; in such a case the outcome is that *all* definitions are renamed. Moreover, since a macro is not recursively expanded and there are no entities other than macros with the old name, a definition for macro '`Old`' cannot refer to '`Old`' in its replacement text. We specify this with a conditional membership: `cmb (name N replText TS) : MacroDef if not N in TS .`

Another assumption is that the old name for the macro is never formed by concatenation in the replacement text of another macro. This could happen if part of the old name is used as an argument of another macro that uses the concatenation operator (`##`) on that argument and some string (like the macro in Fig. 1). Without this assumption, renaming of the old name could be impossible. This check could be added as a precondition to execute the refactoring, as described in [5].

Figure 5 shows the module specifying this refactoring. The operation RenameMacro receives the `Old` name and the `New` name as arguments, and only applies changes to the CFilesMap `FM` if the `New` name is not an existent symbol in the program. Then, the operation renMacroInFiles traverses all files in `FM` and calls renMacroInLS on the line sequence of each file. In turn, renMacroInLS traverses all lines in the LineSeq of a file, calling renMInLine on each line. The latter changes the line it receives as third argument, if that line refers to `Old`, to refer to `New`. Otherwise, the last equation for renMInLine is executed (thanks to the `[owise]` attribute), which does not apply any changes to the line. Last but not least, when there is token sequence involved, renMInLine calls renMacroInTS, which traverses every token in the sequence, replacing `Old` by `New`.

5 Correctness of Refactorings

Based on the formal specification of Cpp and the specification of Cpp's refactorings, this section presents detailed mathematical proofs of the correctness for the refactorings discussed in Sect. 4.

5.1 Correctness of Inline File

Some auxiliary lemmas are first presented, that will help in the proof of correctness of this refactoring.

Lemma 1. *For all variables* `LS1:`*LineSeq*, `LS2:`*LineSeq* *and* `S:`*CppState*,
`state(LS1 ++ LS2, S) = state(LS2, state(LS1, S))`.

```
fmod REN-MACRO-REF is
  --- some module importation
  sort RenameMacroRefactoring .
  subsort RenameMacroRefactoring < CppRefactoring .
  op RenameMacro : Identifier Identifier -> RenameMacroRefactoring .
  op renMacroInFiles : Identifier Identifier CFilesMap -> CFilesMap .
  op renMacroInLS : Identifier Identifier LineSeq -> LineSeq .
  op renMInLine : Identifier Identifier Line -> Line .
  op renMacroInTS : Identifier Identifier TokenSequence -> TokenSequence .
  var FM: CFilesMap. vars Old New I : Identifier. var IdL: IdentifierList. var T: Token.
  vars N F : String. var LS: LineSeq. var L: Line. var TS: TokenSequence.
  ceq FM <- RenameMacro(Old, New) = FM  if New inSet allSymbolNames(FM) .
  eq FM <- RenameMacro(Old, New) = renMacroInFiles(Old, New, FM) [owise] .
  eq renMacroInFiles(Old, New, empty) = empty .
  eq renMacroInFiles(Old, New, [N : LS] FM)
    = [N : renMacroInLS(Old, New, LS)] renMacroInFiles(Old, New, FM) .
  eq renMacroInLS(Old, New, nilLS) = nilLS .
  eq renMacroInLS(Old, New, L LS) = renMInLine(Old,New,L) renMacroInLS(Old,New,LS).
  eq renMInLine(Old, New, #define Old TS cr) = #define New TS cr .
  eq renMInLine(Old, New, #define Old(IdL) TS cr) = #define New(IdL) TS cr .
  ceq renMInLine(Old, New, #define I TS cr)
    = (#define I renMacroInTS(Old, New, TS) cr)   if I =/= Old .
  ceq renMInLine(Old, New, #define I(IdL) TS cr)
    = (#define I(IdL) renMacroInTS(Old, New, TS) cr)   if I =/= Old .
  eq renMInLine(Old, New, #undef Old cr) = #undef New cr .
  eq renMInLine(Old, New, #ifdef Old cr) = #ifdef New cr .
  eq renMInLine(Old, New, #ifndef Old cr) = #ifndef New cr .
  eq renMInLine(Old, New, #if TS cr) = #if renMacroInTS(Old,New,TS) cr .
  eq renMInLine(Old, New, #elif TS cr) = #elif renMacroInTS(Old,New,TS) cr .
  eq renMInLine(Old, New, #include Old cr) = #include New cr .
  eq renMInLine(Old, New, TS cr) = renMacroInTS(Old, New, TS) cr .
  eq renMInLine(Old, New, L) = L     [owise] .
  eq renMacroInTS(Old, New, nil) = nil .
  eq renMacroInTS(Old, New, Old TS) = New renMacroInTS(Old, New, TS) .
  ceq renMacroInTS(Old, New, T TS) = T renMacroInTS(Old, New, TS) if T =/= Old .
endfm
```

Fig. 5. Module implementing Rename Macro refactoring

Proof (by structural induction on `LS1`).
Base Case: `LS1 = nilLS`. The left-hand side of the goal becomes:

```
state(nilLS ++ LS2, S) = state(LS2, S)            by equations for ++
```

The right-hand side becomes:

```
state(LS2, state(nilLS, S) = state(LS2, S)     by equations for state
```

I.H.: The lemma holds for `LS1 = LS`.

Ind. Case: `LS1 = L LS`. The left-hand side of the goal becomes:

```
state((L LS) ++ LS2, S) = state(L (LS++LS2), S)   by equations for ++
= state(LS++LS2, state(L, S))                   by equations for state
```

The right-hand side becomes:

```
state(LS2, state(L LS, S)
= state(LS2, state(LS, state(L, S)))          by equations for state
= state(LS++LS2, state(L, S))                 by I.H.
```

And therefore both the left and right-hand sides of our goal are equal. □

Lemma 2. *For all variables* R:*String, and* LSR,LSX:*LineSeq, such that* LSX *does not contain a line '*#include R cr*', then:* remInclLines(R, LSR, LSX) = LSX.

Proof (by structural induction on LSX).
Base Case: LSX = nilLS.

```
remInclLines(R, LSR, nilLS) = nilLS      by equations for remInclLines
```

Ind.Hyp.: The lemma is true for LSX = LS

Ind. Case: LSX = L:Line LS:LineSeq

```
remInclLines(R, LSR, L LS)
= L remInclLines(R, LSR, LS)             by equations for remInclLines
= L LS                                   by I.H.
```

□

Lemma 3. *For all variables* R,F:*String,* LSR,LSF:*LineSeq,* FM:*CFilesMap, and* S,S':*CppState, such that* LSR *does not contain lines '*#include R cr*' or '*#include F cr*', and if*

```
state(LSR, (files([R:LSR][F:LSF]FM), S)) = files([R:LSR][F:LSF]FM), S'
```

that is, S *is transformed into* S'*, then:*

```
state(LSR, (files([F:remInclLines(R,LSR,LSF)]
             removeIncludes(R,LSR,FM)), S))
= files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM)), S'
```

Proof. This lemma says that if the state operation applied on LSR and the current state transforms S into S', then changing the value of the state attribute files as shown will transform S also into S', i.e., the value of all the state attributes except for files end up with the same value, provided the hypothesis on LSR holds. The proof proceeds by induction on the number of #include directives still to preprocess. Let us call this number NI.

Base Case: NI = 0. The value of the state attribute files is only considered upon a #include directive. If there are none of them, the difference in the contents of files will not interfere with the outcome of the state operation.

I.H. The lemma holds for NI = N.

Ind. Case: NI = N + 1. Let us suppose for convenience that the added #include directive is the first line of LSR. Then LSR = (#include X cr) LS, where X ≠ R and X ≠ F by hypothesis. The goal becomes:

```
state(#include X cr LS, (files([F:remInclLines(R,LSR,LSF)]
             removeIncludes(R,LSR,FM)), S))
= state(LS, state(#include X cr, (files([F:remInclLines(R,LSR,LSF)]
             removeIncludes(R,LSR,FM)), S))  ---by equations for state
= files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM)), S'
                                          ---the goal for this lemma
```

Case: `skip(true)` or `X` is not in `FM`

If the value of the state attribute skip inside `S` is `true`, the inner state operation does not apply any changes to `S` and the lemma holds by I.H. on the outer state operation.

Otherwise, if `X` is not in the files map, then the operation readFile, applied to execute the `#include` directive, returns nilLS, and so the inner state operation does not change the value of `S`. Again the lemma holds by I.H. on the outer state operation.

Case: `skip(false)` and `FM = [X:LSX]FM'`

Let us call `FMg` the value of the state attribute files in our goal. Then:

```
FMg = [F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR, [X:LSX]FM')
    = [F:remInclLines(R,LSR,LSF)] [X:remInclLines(R,LSR,LSX)]
       removeIncludes(R,LSR,FM')     ---by equations for removeIncludes
```

The antecedent of our goal now becomes:

```
state(LS, state(#include X cr, (files(FMg), S)))
= state(LS, state(readFile(X, FMg), (files(FMg), S)))
                                          --- by equations for state
= state(LS, state(remInclLines(R,LSR,LSX), (files(FMg), S)))
                                        --- by equations for readFile
```

Note that `LSX` cannot have a line '`#include R cr`', because `X` was included by `R` and there are no circular references in the structure of the files map. Therefore, we can apply Lemma 2 as follows:

```
state(LS, state(remInclLines(R,LSR,LSX), (files(FMg), S)))
= state(LS, state(LSX, (files(FMg), S)))
```

Since the inner state operation has one less `#include` directive to preprocess, we can apply I.H. to it. Therefore, if:

`state(LSX, (files([R:LSR][F:LSF]FM),S)) = files([R:LSR][F:LSF]FM), S*`

then: `state(LSX, (files(FMg), S)) = files(FMg), S*`.

The goal becomes: `state(LS, (files(FMg, S*))) = files(FMg), S'`

which is true because `LS` has one less `#include` directive to preprocess, so the I.H. is again applicable. □

Theorem 1. *Inlining a file does not change the output of Cpp:*

`preprocess(FM, F) = preprocess(FM <- InlineFile(R), F)`

where `FM` *is a* CFilesMap, `F` *and* `R` *are* Strings, *and when* `R` $\neq$ `F`, `R` *does not come from macro expansion, and all Cpp conditional constructs that are opened inside a file, and only those, are closed in the same file.*

Proof. Note that the hypothesis that all Cpp conditional constructs are opened and closed in the same file is a recommended practice and violations of it are rare.

If the file named `R` is not in the CFilesMap `FM`, then no changes are applied and the theorem is true. Similarly, if `F` is not the name of a file in `FM`, the preprocess operation returns nil in both sides. Otherwise, `FM = ([R:LSR][F:LSF]FM')`,

where LSR and LSF are the LineSeqs corresponding to R and F respectively, and FM' is the rest of the CFilesMap. Applying the first and fourth equations in module INLINE-FILE-REF the goal becomes:

```
preprocess([R:LSR][F:LSF]FM', F)
= preprocess([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM'), F)
```

and applying the equation for preprocess on both sides:

```
returnOutput(state(LSF, (files([R:LSR][F:LSF]FM'), S)))
= returnOutput(state(remInclLines(R,LSR,LSF),
    (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S)))
```

The proof then proceeds by structural induction on LSF.

Base case: LSF = nilLS.

Applying the operation remInclLines(R,LSR,nilLS) returns nilLS so the goal becomes:

```
returnOutput(state(nilLS, (files([R:LSR][F:nilLS]FM'),
                          outputStream(nil), S)))
= returnOutput(state(nilLS,
    (files([F:nilLS] removeIncludes(R,LSR,FM')),outputStream(nil), S)))
```

which returns nil on both sides and therefore holds.

I.H.: The theorem is true for LSF = LSa, i.e.:

```
returnOutput(state(LSa, (files([R:LSR][F:LSa]FM'), S)))
= returnOutput(state(remInclLines(R,LSR,LSa),
    (files([F:remInclLines(R,LSR,LSa)] removeIncludes(R,LSR,FM')), S)))
```

Inductive case: LSF = L LSa
(Note that we append a line L at the beginning to make equations applicable on this first line, but we assume a non-initial starting state). The left-hand side of our goal becomes:

```
returnOutput(state(L LSa, (files([R:LSR][F:LSF]FM'), S)))
= returnOutput(state(LSa, state(L,(files([R:LSR][F:LSF]FM'), S))))
                                              ---by equations for state
```

Case 1. L = #include R cr.

The left-hand side of the goal becomes:

```
returnOutput(state(LSa,
          state(#include R cr,(files([R:LSR][F:LSF]FM'), S))))
= returnOutput(state(LSa, state(LSR, (files([R:LSR][F:LSF]FM'), S))))
                                  ---by equations for state and readFile
```

The right-hand side becomes:

```
returnOutput(state(remInclLines(R,LSR,(#include R cr) LSa),
    (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S)))
= returnOutput(state(LSR ++ remInclLines(R,LSR,LSa),
    (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S)))
                                    ---by equations for remInclLines
= returnOutput(state(remInclLines(R,LSR,LSa), state(LSR,
   (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S))))
                                    ---by Lemma 1
```

The goal now becomes:

```
returnOutput(state(LSa, state(LSR, (files([R:LSR][F:LSF]FM'), S))))
= returnOutput(state(remInclLines(R,LSR,LSa), state(LSR,
   (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S))))
```

Applying Lemma 3 to the inner state operation results in:

```
returnOutput(state(LSa, (files([R:LSR][F:LSF]FM'), S')))
= returnOutput(state(remInclLines(R,LSR,LSa),
   (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S')))
```

which is true by I.H.

Case 2. `L` $\neq$ `#include R cr`.
The right-hand side becomes:

```
returnOutput(state(remInclLines(R, LSR, L LSa),
    (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S)))
= returnOutput(state(L remInclLines(R,LSR,LSa),
    (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S)))
                                    ---by equations for remInclLines
= returnOutput(state(remInclLines(R,LSR,LSa), state(L,
   (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S))))
                                    ---by equations for state
```

The goal now becomes:

```
returnOutput(state(LSa, state(L,(files([R:LSR][F:LSF]FM'), S))))
= returnOutput(state(remInclLines(R,LSR,LSa), state(L,
   (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S))))
```

Note that `L` is not a `#include R cr` line by hypothesis, nor a `#include F cr` line, because `L` is a line of `LSF`, and circular references are not allowed in the CFilesMap. Therefore, the hypothesis of Lemma 3 applies for the inner state operation, having `L` as a LineSeq with a single element. Applying Lemma 3 to the inner state operation the goal becomes:

```
returnOutput(state(LSa, (files([R:LSR] [F:LSF] FM'), S')))
= returnOutput(state(remInclLines(R,LSR,LSa),
   (files([F:remInclLines(R,LSR,LSF)] removeIncludes(R,LSR,FM')), S')))
```

which is true by I.H. □

5.2 Correctness of Extract File

Theorem 2. *Extracting a file does not change the output of Cpp:*
`preprocess(FM,B) = preprocess(FM <- ExtractFile(F,LS2,NewF), B)`
where `FM:` *CFilesMap,* `B,F,NewF:` *String, and* `B` *may or may not be the same as* `F`.

Proof. The proof of this theorem is very similar to the proof of Inline File, and also proceeds by induction on the line sequence of `B`, the starting file for preprocessing. Moreover, InlineFile and ExtractFile may be considered inverse operations, hence the similarity of the proof. Nevertheless, the proof of ExtractFile is easier, since it only involves changes in the line sequence of a single file (`F`). We were able to prove this theorem automatically with the Maude Inductive Theorem Prover (ITP) [19], for the case that the extracted line sequence does not contain Cpp directives.

5.3 Correctness of Rename Macro

The following three lemmas are used in the proof of Theorem 3.

Lemma 4. *For all variables* `Old,New:` *Identifier and* `TS:` *TokenSequence, such that* `TS` *does not contain the token* `Old`*, then:*
`renMacroInTS(Old, New, TS) = TS`.

Proof (by structural induction on TS).
Base Case: `TS = nil`.

```
renMacroInTS(Old, New, nil) = nil       by equations for renMacroInTS
```

I.H.: The lemma is true for `TS = TSb`

Ind. Case: `TS = T:Token TSb:TokenSequence`

```
renMacroInTS(Old, New, T TSb)
= T renMacroInTS(Old, New, TSb)          by equations for renMacroInTS
= T TSb                                  by I.H.
```

□

Lemma 5. *For all variables* `Old,New:` *Identifier and* `TS1,TS2:` *TokenSequences,*

```
renMacroInTS(Old, New, TS1 TS2)
= renMacroInTS(Old, New, TS1) renMacroInTS(Old, New, TS2)
```

Proof (by structural induction on TS1).
Base Case: `TS1 = nil`. The base case holds trivially since `(nil TS2) = TS2` by property of a TokenSequence.

I.H.: The lemma is true for `TS1 = TSb`

Ind. Case: `TS1 = T:Token TSb:TokenSequence`.
Case `T` $\neq$ `Old`.

```
renMacroInTS(Old, New, T TSb TS2)
= T renMacroInTS(Old, New, TSb TS2)       by equations for renMacroInTS
= T renMacroInTS(Old,New,TSb) renMacroInTS(Old,New,TS2)         by I.H.
= renMacroInTS(Old,New,T TSb) renMacroInTS(Old,New,TS2)
                                          by equations for renMacroInTS
```

The case when `T = Old` follows similarly. □

Lemma 6. *Given variables* `L`: *Line,* `FM,FMg`: *CFilesMap,* `O,O*`: *OutputStream,* `MT,MT',MTg,MTg'`: *MacroTable and* `S,S'`: *CppState, and if:*

```
(a) state(L, (files(FM),macroTbl(MT),outputStream(O),S)
     = files(FM),macroTbl(MT'),outputStream(O*),S'
```

then

```
(b) state(renMInLine(L), (files(FMg),macroTbl(MTg),outputStream(O),S)
    = files(FMg),macroTbl(MTg'),outputStream(O*),S'
```

when the difference between `FM` *and* `FMg` *and between* `MT` *and* `MTg`*, are only those caused by the application of* `RenameMacro(Old,New)`*, and when, as stated in the previous section,* `Old` *is only used as a macro name, and its name is not computed through macro concatenation.*

Proof. The proof proceeds by case analysis on the kind of Line that `L` can be.

Case 1. `L = #define Old TS cr`, i.e., a definition for macro `Old`, where `TS`: TokenSequence. Note that, for simplicity, we consider the case when the macro does not have parameters, but in fact parameters do not make any difference in the proof. The refactored version of that line is:

```
renMInLine(#define Old TS cr) = #define New TS cr
```

If the value of the attribute skip in the state `S` is `true`, then the state operation does not change `S`. If the value of the state attribute skip is false, the operation state is applied to `(a)` as follows:

```
state(#define Old TS cr, (files(FM),macroTbl(MT),outputStream(O), S))
= files(FM),macroTbl([Old:(name Old replText TS)]MT),outputStream(O),S.
```

The operation state applied to `(b)` will be:

```
state(#define New TS cr, (files(FMg),macroTbl(MTg),outputStream(O), S))
=files(FMg),macroTbl([New:(name New replText TS)]MTg),outputStream(O),S.
```

Since the input line is a macro definition in both `(a)` and `(b)`, the state operation does not change the value of outputStream, which is equal in both cases. The difference in the macro table will affect Case 6, when the macro is actually called.

Case 2. `L = #define I TS1 Old TS2 cr`, where `I` $\neq$ `Old` and `TS1` and `TS2` are TokenSequences, that is, `L` is a definition for a macro `I` that refers to `Old` in its replacement text. Applying `renMInLine(L)` results in:

```
#define I TS1 New TS2 cr
```

Similarly to Case 1, applying the operation state to `L` and its refactored version will not change the value of the outputStream attribute. The difference will appear in the entry for `I` in the macro table, which will only affect calls to the macro (Case 6).

Case 3. `L = #undef Old cr`, i.e., an un-definition for `Old`. The operation renMInLine is applied to `L` as follows:

```
renMInLine(#undef Old cr) = #undef New cr
```

Case 3.1. `Old` was previously defined as a macro.
In this case, we know by Case 1 that the starting state in `(a)` has the macro table looking as follows: `macroTbl(MTb [Old:(name Old replText TS)])`.
Therefore the equation `(a)` looks like:

```
state(#undef Old cr, (files(FM),
  macroTbl(MTb [Old:(name Old replText TS)]),outputStream(O),S))
= files(FM), macroTbl(MT), outputStream(O), S .
```

The starting state in `(b)` will have the macro table as follows:

```
macroTbl(MTgb [New:(name New replText TS)]).
```

The equation `(b)` looks as follows:

```
state(#undef New cr, (files(FMg),
  macroTbl(MTgb [New:(name New replText TS)]), outputStream(O), S))
= files(FMg), macroTbl(MTgb), outputStream(O), S .
```

Therefore, the resulting outputStream is the same in both cases.

Case 3.2. `Old` was not previously defined as a macro.
In this case, the macro tables will look the same on both sides, and the `#undef` directive will not cause any changes to the macro table or to any other state attribute.

Case 4. `L = #ifdef Old cr`, a conditional directive involving `Old`. Applying `renMInLine(Old,New,L)` results in:

```
#ifdef New cr
```

This case follows easily from observing that, if `Old` was previously defined as a macro in the original program, then `New` will be defined as a macro in the refactored program. Similarly, if `Old` was not defined, `New` will not be defined either. Therefore, the result of applying the state operation will be the same on `L` and on `renMInLine(L)`. The case for other conditional directives follows similarly.

Case 5. `L = #include Old cr`, i.e., a computed include line. In this case, `Old` must be previously defined as a macro, and expand to a string that represents a file name. In the refactored program, the definition for `Old` has been renamed to `New` but expands to the same file name (see Case 1). Therefore, the state operation applied to `L` or its refactored version (`#include New cr`) yields the same result.

Case 6. `L = TS1 Old TS2 cr`, i.e., a line with a token sequence that includes `Old` as a token, where `TS1` and `TS2` are TokenSequences that do not refer to `Old` (not even indirectly through macro expansion). This line may be the result of previous macro expansions. The refactored version of that line is:

```
renMInLine(Old, New, TS1 Old TS2 cr)
= renMacroInTS(Old, New, TS1 Old TS2) cr
                                          ---by equations for renMInLine
= renMacroInTS(Old,New,TS1) renMacroInTS(Old,New,Old TS2) cr
                                                          ---by Lemma 5
= TS1 renMacroInTS(Old,New, Old TS2) cr                   ---by Lemma 4
= TS1 New renMacroInTS(Old,New,TS2) cr
                                       ---by equations for renMacroInTS
= TS1 New TS2 cr                                          ---by Lemma 4
```

Note that, by the assumptions in the hypothesis, `Old` must be previously defined as a macro. Similarly to Case 3.1, there exists a definition for `Old` before the current line. The macro table of the un-refactored program, before preprocessing the current line is therefore: `macroTbl(MT [Old:(name Old replText TS)])`.

The operation state is applied to `L` and the current state as follows:

```
state(TS1 Old TS2 cr, (files(FM),
  macroTbl(MTb [Old:(name Old replText TS)]), outputStream(O), S))
```

which is further decomposed into several applications of the state operation, one for each token in `TS1`, one for the token `Old`, and one for each token in `TS2`, resulting in:

```
files(FM), macroTbl(MT [Old:(name Old replText TS)]),
    outputStream(O TS1' TS TS2'), S .
```

where `TS1'` is the result of preprocessing `TS1`, `TS` is the expansion of `Old`, and `TS2'` is the result of preprocessing `TS2`.

In the refactored program, the previous '`#define Old TS cr`' has been transformed by the refactoring to a line '`#define New TS cr`', so the macro table would look like: `macroTbl(MT [New:(name New replText TS)])`.

The operation state is applied to `renMInLine(L)` and the current state as follows:

```
state(TS1 New TS2 cr, (files(FMg),
 macroTbl(MT [New:(name New replText TS)]), outputStream(O), S))
```

which is further decomposed into several applications of the state operation, one for each token in TS1, one for the token New, and one for each token in TS2, yielding:

```
files(FMg), macroTbl(MT [New:(name New replText TS)]),
   outputStream(O TS1' TS TS2'), S .
```

Since by hypothesis we have assumed that TS1 and TS2 did not refer to Old directly or indirectly, the result of preprocessing them is the same that in the un-refactored program, i.e. TS1' and TS2' respectively. On the other hand, the expansion of New is the same than for Old, because the refactoring only changed the name of the macro, but not its replacement text. Therefore, the resulting output is the same in both cases.

Case 7. L is something different from the previous six cases. In this case, the equation for renMInLine that applies is the last one, that is:

```
eq renMInLine(Old,New,L) = L [owise]
```

because of the [owise] attribute, so no changes are applied to L and the lemma trivially holds. As already mentioned, the [owise] attribute is syntactic sugar for a standard conditional specification [6].

□

Theorem 3. *Renaming a macro does not change the output of Cpp:*

```
preprocess(FM, F) = preprocess(FM<-RenameMacro(Old, New), F)
```

where FM *is a* CFilesMap*,* F *is a* String*, and* Old *and* New *are* Identifiers*, and when* Old *is only used as a macro name, and its name is not computed through macro concatenation.*

Proof. If New is not a new symbol, i.e., if it appears as a symbol in the program, the refactoring is not applied and the theorem trivially holds. Similarly, if F is not the name of a file in FM, the preprocess operation returns nil in both sides. Otherwise, FM = [F:LSF] FM', where LSF is the LineSeq corresponding to F and FM' is the rest of the CFilesMap. Applying the equations in module REN-MACRO-REF the goal becomes:

```
preprocess([F:LSF]FM', F)
= preprocess([F:renMacroInLS(Old,New,LSF)]
              renMacroInFiles(Old,New,FM'), F)
```

Let FMg = [F:renMacroInLS(Old,New,LSF)]renMacroInFiles(Old,New,FM'). Applying the equation for preprocess on both sides:

```
returnOutput(state(LSF, (files(FM), S)))
= returnOutput(state(renMacroInLS(Old,New,LSF), (files(FMg), S)))
```

The proof proceeds by structural induction on LSF.

Base case: LSF = nilLS.

The base case holds trivially, since renMacroInLS(Old,New,nilLS)=nilLS and the operation state does not apply any changes when the first parameter is the empty token sequence.

I.H.: The theorem is true for `LSF = LSa`, i.e.:

```
returnOutput(state(LSa, (files(FM), macroTbl(MT), S)))
= returnOutput(state(renMacroInLS(Old,New,LSa),
                     (files(FMg), macroTbl(MTg), S)))
```

where the difference between `MT` and `MTg` is caused by the application of the rules of `RenameMacro(Old,New)`, i.e., where `MT` has an entry for `Old`, `MTg` has an entry for `New`, and in every entry in `MT` where the replacement text refers to `Old`, in `MTg` it refers to `New`.

Inductive case: `LSF = L LSa`
(Note that we append a line `L` at the beginning to make equations applicable on this first line, but we assume a non-initial starting state).

The left-hand side of our goal becomes:

```
returnOutput(state(L LSa, (files(FM), macroTbl(MT), S)))
= returnOutput(state(LSa, state(L,(files(FM), macroTbl(MT), S))))
                                                ---by equations for state
= returnOuput(state(LSa, (files(FM), macroTbl(MT'), S')))
```

where we assume that applying the `state` operation on `L` transforms `MT` into `MT'` and `S` into `S'`. The right-hand side of the goal becomes:

```
returnOutput(state(renMacroInLS(Old,New,L LSa),
          (files(FMg),macroTbl(MTg),S)))
= returnOutput(state(renMInLine(Old,New,L) renMacroInLS(Old,New,LSa),
          (files(FMg),macroTbl(MTg),S)))
                                      ---by equations for renMacroInLS
= returnOutput(state(renMacroInLS(Old,New,LSa),
          state(renMInLine(Old,New,L), (files(FMg),macroTbl(MTg),S))))
                                                ---by equations for state
= returnOutput(state(renMacroInLS(Old,New,LSa),
      (files(FMg, macroTbl(MTg'), S'))))                  ---by Lemma 6
= returnOutput(state(LSa, (files(FM),macroTbl(MTg'),S'))) ---by I.H.
```

which concludes the theorem. □

6 Conclusion

We have presented an executable algebraic semantics of Cpp and of two Cpp refactorings in Maude. To the best of our knowledge, these formal specifications are the first ever written for Cpp and Cpp refactorings. We have used them to prove that these Cpp refactorings preserve program behavior. Moreover, the independence of Cpp from the underlying programming language makes our results generally applicable not just to C, but to any other language that uses Cpp (like C++ or Fortran).

Future work includes specifying other refactorings on Cpp directives and possibly mechanize their proofs of correctness. We have started to work in this

direction using ITP, the Maude Inductive Theorem Prover [19], and we have proven parts of some theorems under more restrictive assumptions [20].

Future plans also include applying the same methodology used in this paper, i.e., semantic-based correctness proofs, to the C programming language itself. A specification for C is more complicated than a specification for Cpp, but once we have it, we will be able to combine it with the specification for Cpp and use this combined specification to prove the correctness of refactorings on both C and Cpp. These refactorings will be harder to implement, both because the specifications are more complicated and because the refactorings will require context information, but we believe that Maude and the tools developed for it will help as in the process.

Acknowledgements. We thank Ralf Sasse, Mark Hills and Manuel Clavel for their help in this project. This research has been supported in part by ONR Grant N00014-02-1-0715.

References

1. Fowler, M.: Refactoring. Improving the Design of Existing Code. Addison-Wesley (1999)
2. Arnold, R.: Software restructuring. In: Proceedings IEEE. Volume 77. (1989)
3. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
4. Ernst, M., Badros, G., Notkin, D.: An empirical analysis of C preprocessor use. Revision of Technical Report UW-CSE-97-04-06, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle (1999)
5. Garrido, A.: Program Refactoring in the Presence of Preprocessor Directives. PhD thesis, University of Illinois at Urbana-Champaign. (2005)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Olliet, N., Meseguer, J., Talcott, C.: Maude Manual (Ver. 2.2), http://maude.cs.uiuc.edu/maude2-manual/. (2005)
7. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, ed., Proc. WADT'97. Springer LNCS 1376. (1998)
8. Visser, E.: A survey of strategies in rule-based program transformation systems. Journal of Symbolic Computation **40**(1) (2005)
9. Visser, E.: Program transformation with Stratego/TX: rules, strategies, tools, and systems in StrategoXT-0.9. In: Domain-Specific Program Generation. In: LNCS vol.3016. Springer-Verlag. (2004)
10. van den Brand, M., Klint, P., Vinju, J.: Term rewriting with traversal functions. (ACM Transaction on Software Engineering and Methodology)
11. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming **52**(1-3) (2004)
12. Farzan, A., Chen, F., , Meseguer, J., Roşu, G.: Formal Analysis of Java Programs in JavaFAN. In: Int. Conf. on Computer Aided Verification (CAV'04), Boston, Mass. (2004)
13. Sasse, R.: Tackets vs. rewriting logic - relating semantics of Java. Master's thesis, University of Karlsruhe (2005)

14. Ahrendt, W., Roth, A., Sasse, R.: Automatic validation of transformation rules for Java verification against a rewriting semantics. In: Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'05), Jamaica (2005)
15. Harbison, S.P., Jr., G.L.S.: C. A Reference Manual. Third edn. Prentice Hall (1991)
16. Stallman, R., Weinberg, Z.: The C preprocessor. GNU Online documentation. http://gcc.gnu.org/onlinedocs/ (2001)
17. José Meseguer and Grigore Roşu: The rewriting logic semantics project. In: Proc. of Structural Operational Semantics (SOS'05). (2005)
18. Roşu, G.: CS422 - Programming Language Design - Class Notes. http://fsl.cs.uiuc.edu/ grosu/classes/2004/fall/cs422/ (2004)
19. Clavel, M., Palomino, M.: The ITP Tool Manual. Universidad Complutense, Madrid, http://maude.sip.ucm.es/itp/. (2005)
20. Garrido, A., Meseguer, J., Johnson, R.: Algebraic Semantics of the C Preprocessor and Correctness of its Refactorings. Technical Report UIUCDCS-R-2006-2688, Dept. of Computer Science. Univ. of Illinois at Urbana-Champaign. https://netfiles.uiuc.edu/garrido/www/publications.html (2006)

Appendix

The following is the Maude specification of Cpp.

```
--- ------------- ---
--- SYNTAX OF CPP ---
--- ------------- ---
fmod IDENTIFIER is
  pr QID .
  sort Identifier IdentifierList IdentifierListP .
  subsorts Qid < Identifier < IdentifierList .
  op `(` : -> IdentifierListP .
  op `(_` : IdentifierList -> IdentifierListP .
  op _,_ : IdentifierList IdentifierList -> IdentifierList [assoc] .
  op size : IdentifierListP -> Nat .
  op _in_ : Identifier IdentifierListP -> Bool .
  op pos : Identifier IdentifierListP -> Nat .
  op cons : Identifier IdentifierListP -> IdentifierListP .
  vars I I' : Identifier . var IL : IdentifierList . var N : Nat .
  var ILP : IdentifierListP .
  eq size(()) = 0 .
  eq size((I, IL)) = 1 + size((IL)) .
  eq size((I)) = 1 .
  eq I in () = false .
  eq I in (I', IL) = (I == I') or (I in (IL)) .
  eq I in (I') = (I == I') .
  eq pos(I, ()) = 0 .
  ceq pos(I, (I', IL)) = 1 if (I == I') .
  ceq pos(I, (I', IL)) = 1 + pos(I, (IL)) if (I =/= I') and (I in (IL)) .
  ceq pos(I,(I')) = 1 if (I == I') .
  eq pos(I, (IL)) = 0 [owise] .
  eq cons(I,()) = (I) .
  eq cons(I,(IL)) = (I,IL) .
endfm

fmod TOKEN is
  pr IDENTIFIER .
  sorts Token TokenSequence .
  subsort Identifier < Token < TokenSequence .
  op nil : -> TokenSequence [ctor] .
  op __ : TokenSequence TokenSequence -> TokenSequence [ctor assoc id: nil] .
```

```
  op _inTS_ : Token TokenSequence -> Bool .
  vars T T' : Token . var TS : TokenSequence .
  eq T inTS nil = false .
  eq T inTS (T' TS) = (T == T') or (T inTS TS) .
endfm

fmod COND-EXP-SYNTAX is
  pr TOKEN . pr INT .
  sort CondExp .
  subsort Identifier < CondExp .
  op e : Int -> CondExp .
endfm

fmod DEF-COND-SYNTAX is ex COND-EXP-SYNTAX .
  op defined_ : Identifier -> CondExp .
endfm

fmod ARITH-EXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _+_ : CondExp CondExp -> CondExp [prec 40 gather(e E)] .
  op _-_ : CondExp CondExp -> CondExp [prec 40 gather(e E)] .
  op _*_ : CondExp CondExp -> CondExp [prec 35 gather(e E)] .
  op _/_ : CondExp CondExp -> CondExp [prec 35 gather(e E)] .
  op _%_ : CondExp CondExp -> CondExp [prec 35 gather(e E)] .
endfm

fmod BIT-EXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _<<_ : CondExp CondExp -> CondExp [prec 42] .
  op _>>_ : CondExp CondExp -> CondExp [prec 42] .
  op _&_ : CondExp CondExp -> CondExp [prec 46] .
  op _^_ : CondExp CondExp -> CondExp [prec 46] .
  op _|_ : CondExp CondExp -> CondExp [prec 46] .
endfm

fmod REXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _<_  : CondExp CondExp -> CondExp [prec 44] .
  op _<=_ : CondExp CondExp -> CondExp [prec 44] .
  op _>_  : CondExp CondExp -> CondExp [prec 44] .
  op _>=_ : CondExp CondExp -> CondExp [prec 44] .
  op _==_ : CondExp CondExp -> CondExp [prec 45] .
  op _!=_ : CondExp CondExp -> CondExp [prec 45] .
endfm

fmod BEXP-SYNTAX is ex COND-EXP-SYNTAX .
  op !_ : CondExp -> CondExp [prec 30] .
  op _&&_ : CondExp CondExp -> CondExp [prec 49] .
  op _||_ : CondExp CondExp -> CondExp [prec 51] .
endfm

fmod CEXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _?_:_ : CondExp CondExp CondExp -> CondExp [prec 55] .
endfm

fmod ALL-COND-EXP-SYNTAX is
  pr COND-EXP-SYNTAX .
  pr DEF-COND-SYNTAX .
  pr ARITH-EXP-SYNTAX .
  pr BIT-EXP-SYNTAX .
  pr REXP-SYNTAX .
  pr BEXP-SYNTAX .
  pr CEXP-SYNTAX .
endfm

fmod CPP-DIR-SYNTAX is
  sort CppDirective .
endfm

fmod DEFINE-SYNTAX is ex CPP-DIR-SYNTAX .
  pr TOKEN .
```

```
  sorts MacroDefDir MacroUndefDir .
  subsort MacroDefDir MacroUndefDir < CppDirective .
  op #define__cr : Identifier TokenSequence -> MacroDefDir .
  op #define___cr : Identifier IdentifierListP TokenSequence -> MacroDefDir .
  op #undef_cr : Identifier -> MacroUndefDir .
endfm

fmod INCLUDE-SYNTAX is ex CPP-DIR-SYNTAX .
  pr IDENTIFIER .
  sorts IncludeDir FileName .
  subsort IncludeDir < CppDirective .
  op #include_cr : String -> IncludeDir .
  op #include_cr : Identifier -> IncludeDir .
endfm

fmod COND-DIR-SYNTAX is ex CPP-DIR-SYNTAX .
  pr TOKEN . pr ALL-COND-EXP-SYNTAX .
  sort CondDir .
  subsort CondDir < CppDirective .
  op #if_cr : TokenSequence -> CondDir .
  op #ifdef_cr : Identifier -> CondDir .
  op #ifndef_cr : Identifier -> CondDir .
  op #elif_cr : TokenSequence -> CondDir .
  op #else`cr : -> CondDir .
  op #endif`cr : -> CondDir .
endfm

fmod LINE-SEQ-SYNTAX is
  pr CPP-DIR-SYNTAX .
  pr TOKEN .
  sorts Line LineSeq .
  subsorts CppDirective < Line .
  op _cr : TokenSequence -> Line .
  op nilLS : -> LineSeq [ctor] .
  op __ : Line LineSeq -> LineSeq [ctor] .
  op _++_ : LineSeq LineSeq -> LineSeq .
  op _inLS_ : LineSeq LineSeq -> Bool .
  op _consecInLS_ : LineSeq LineSeq -> Bool .
  vars L L' : Line . vars LS1 LS2 : LineSeq .
  eq nilLS ++ LS2 = LS2 .
  eq (L LS1) ++ LS2 = L (LS1 ++ LS2) .
  eq nilLS inLS LS2 = true .
  eq L LS1 inLS nilLS = false .
  eq L LS1 inLS L LS2 = LS1 consecInLS LS2 .
  ceq L LS1 inLS L' LS2 = L LS1 inLS LS2 if L =/= L' .
  eq nilLS consecInLS LS2 = true .
  eq L LS1 consecInLS nilLS = false .
  eq L LS1 consecInLS L LS2 = LS1 consecInLS LS2 .
  eq L LS1 consecInLS L' LS2 = false .
endfm

fmod CPP-SYNTAX is
  pr ALL-COND-EXP-SYNTAX .
  pr DEFINE-SYNTAX .
  pr INCLUDE-SYNTAX .
  pr COND-DIR-SYNTAX .
  pr LINE-SEQ-SYNTAX .
endfm

--- ---------------- ---
--- SEMANTICS OF CPP ---
--- ---------------- ---
fmod TOKEN-TO-ARG is pr TOKEN .
  sort TokenSeqList .
  subsort TokenSequence < TokenSeqList .
  op nilTSL : -> TokenSeqList .
  op _;_ : TokenSeqList TokenSeqList -> TokenSeqList [assoc id: nilTSL] .
  op size : TokenSeqList -> Nat .
```

```
  op elemAtTS : Nat TokenSeqList -> TokenSequence .
  op toTokenSeqList : TokenSequence -> TokenSeqList .
  var T : Token . vars TS1 TS2 TSA : TokenSequence .
  var TSL : TokenSeqList . var N : Nat .
  eq size(nilTSL) = 0 .
  eq size(TS1 ; TSL) = 1 + size(TSL) .
  eq elemAtTS(1, (TS1 ; TSL)) = TS1 .
  eq elemAtTS(s(N), (TS1 ; TSL)) = elemAtTS(N, TSL) .
  ceq toTokenSeqList(TS1) = TS1 if not (’‘, inTS TS1) .
  ceq toTokenSeqList(TS1 ’‘, TS2) = TS1 ; toTokenSeqList(TS2)
    if not (’‘, inTS TS1) and (’‘, inTS TS2) .
  ceq toTokenSeqList(TS1 ’‘, TS2) = TS1 ; TS2
    if not (’‘, inTS TS1) and not (’‘, inTS TS2) .
endfm

fmod MACRO-DEF is
  pr TOKEN . pr STRING . pr TOKEN-TO-ARG .
  sort MacroDef .
  op name_replText_ : Identifier TokenSequence -> MacroDef .
  op name_params_replText_ : Identifier IdentifierListP TokenSequence -> MacroDef .
  op name : MacroDef -> Identifier .
  op hasArgs : MacroDef -> Bool .
  op expand : MacroDef -> TokenSequence .
  op expandWithArgs : MacroDef TokenSeqList -> TokenSequence .
  op ex-rec : IdentifierListP TokenSequence TokenSeqList -> TokenSequence .
  op dquote : -> Qid .

  var N : Identifier . var TS : TokenSequence . vars T T2 : Token .
  var PL : IdentifierListP .
  eq name(name N replText TS) = N .
  eq name(name N params PL replText TS) = N .
  eq hasArgs(name N replText TS) = false .
  eq hasArgs(name N params PL replText TS) = true .
  eq expand(name N replText TS) = ex-rec(‘(‘), TS, nilTSL) .
  var TSL : TokenSeqList .
  ceq expandWithArgs(name N params PL replText TS, TSL) = nil
    if ( size(PL)  =/= size(TSL) ) .
  eq expandWithArgs(name N params PL replText TS, TSL)
    = ex-rec(PL, TS, TSL) [owise] .
  eq ex-rec(PL, nil, TSL) = nil .
  eq ex-rec(PL, ’# T TS, TSL) = dquote elemAtTS(pos(T, PL), TSL) dquote
                       ex-rec(PL, TS, TSL) .
  ceq ex-rec(PL, T ’## T2 TS, TSL)
    = qid(string(T) + string(T2)) ex-rec(PL, TS, TSL)
    if not(T in PL) and not(T2 in PL) .
  ceq ex-rec(PL, T ’## T2 TS, TSL)
    = qid(string(elemAtTS(pos(T, PL), TSL)) + string(T2)) ex-rec(PL, TS, TSL)
    if (T in PL) and not(T2 in PL) .
  ceq ex-rec(PL, T ’## T2 TS, TSL)
    = qid(string(T) + string(elemAtTS(pos(T2, PL), TSL))) ex-rec(PL, TS, TSL)
    if not(T in PL) and (T2 in PL) .
  eq ex-rec(PL, T ’## T2 TS, TSL) = qid(string(elemAtTS(pos(T, PL), TSL)) +
       string(elemAtTS(pos(T2, PL), TSL))) ex-rec(PL, TS, TSL) [owise] .
  ceq ex-rec(PL, T TS, TSL) = T ex-rec(PL, TS, TSL) if not(T in PL) .
  ceq ex-rec(PL, T TS, TSL) = elemAtTS(pos(T, PL), TSL) ex-rec(PL, TS, TSL)
    if (T in PL) .
endfm

fmod MACRO-TABLE is
  pr MACRO-DEF .
  sort MacroTable .
  op empty : -> MacroTable .
  op [_:_] : Identifier MacroDef -> MacroTable .
  op __ : MacroTable MacroTable -> MacroTable [assoc comm id: empty] .
  op _[_] : MacroTable Identifier -> MacroDef .
  op _[_<-_] : MacroTable Identifier MacroDef -> MacroTable .
  op isMacro : Identifier MacroTable -> Bool .
  op isMacroWithArgs : Identifier MacroTable -> Bool .
```

```
  op isMacroWithoutArgs : Identifier MacroTable -> Bool .
  op remove : Identifier MacroTable -> MacroTable .
  vars N N' : Identifier .  vars M M' : MacroDef . var MT : MacroTable .
  eq ([N : M] MT)[N] = M .
  eq ([N : M'] MT)[N <- M] = [N : M] MT .
  eq MT[N <- M] = MT [N : M] [owise] .
  eq isMacro(N, empty) = false .
  eq isMacro(N, ([N' : M] MT)) = (N == N') or isMacro(N, MT) .
  eq isMacroWithArgs(N, MT) = isMacro(N, MT) and hasArgs(MT[N]) .
  eq isMacroWithoutArgs(N, MT) = isMacro(N, MT) and not hasArgs(MT[N]) .
  eq remove(N, empty) = empty .
  eq remove(N, ([N : M] MT)) = remove(N, MT) .
  ceq remove(N, ([N' : M] MT)) = [N' : M] remove(N, MT) if N =/= N' .
endfm

fmod COND-AUX is
  pr STRING . pr RAT . pr CONVERSION .
  op isNumber : String -> Bool .
  op toInt : String -> Int .
  var S : String .
  eq toInt(S) = rat(S, 10) .
endfm

fmod COND-EXP-SEMANTICS is pr COND-AUX .
  pr COND-EXP-SYNTAX . pr MACRO-TABLE .
  op evalB : CondExp MacroTable -> Bool .
  op evalA : CondExp MacroTable -> Int .
  op toCondExp : TokenSequence MacroTable -> CondExp .
  var MT : MacroTable . var X : Int . var T : Token . var AS : TokenSequence .
  ceq evalB(e(X), MT) = true if X =/= 0 .
  eq evalB(e(0), MT) = false .
  eq evalA(e(X), MT) = X .
  eq evalB(T, MT) = false .     --- T is not a macro
  eq evalA(T, MT) = 0 .     --- T is not a macro
  ceq toCondExp(T, MT) = toCondExp(expand(MT[T]), MT)
    if isMacroWithoutArgs(T, MT) .
  ceq toCondExp(T '`( AS '`), MT)
    = toCondExp(expandWithArgs(MT[T], toTokenSeqList(AS)), MT)
    if isMacroWithArgs(T, MT) .
  ceq toCondExp(T, MT) = e(toInt(string(T)))  if isNumber(string(T)) .
endfm

fmod DEF-COND-SEMANTICS is pr DEF-COND-SYNTAX .
  ex COND-EXP-SEMANTICS .
  var N : Identifier . var MT : MacroTable .
  eq evalB(defined N, MT) = isMacro(N, MT) .
  eq toCondExp('defined N, MT) = defined N .
endfm

fmod ARITH-EXP-SEMANTICS is pr ARITH-EXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
  eq evalA(E + E', MT) = evalA(E, MT) + evalA(E', MT) .
  eq evalA(E - E', MT) = evalA(E, MT) - evalA(E', MT) .
  eq evalA(E * E', MT) = evalA(E, MT) * evalA(E', MT) .
  eq evalA(E / E', MT) = evalA(E, MT) quo evalA(E', MT) .
  eq evalA(E % E', MT) = evalA(E, MT) rem evalA(E', MT) .
  var T : Token . var TS : TokenSequence .
  eq toCondExp(T '+ TS, MT) = toCondExp(T, MT) + toCondExp(TS, MT) .
  eq toCondExp(T '- TS, MT) = toCondExp(T, MT) - toCondExp(TS, MT) .
  eq toCondExp(T '* TS, MT) = toCondExp(T, MT) * toCondExp(TS, MT) .
  eq toCondExp(T '/ TS, MT) = toCondExp(T, MT) / toCondExp(TS, MT) .
  eq toCondExp(T '% TS, MT) = toCondExp(T, MT) % toCondExp(TS, MT) .
endfm

fmod BIT-EXP-SEMANTICS is pr BIT-EXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
```

```
  eq evalA(E << E', MT) = evalA(E, MT) << evalA(E', MT) .
  eq evalA(E >> E', MT) = evalA(E, MT) >> evalA(E', MT) .
  eq evalA(E & E', MT) = evalA(E, MT) & evalA(E', MT) .
  eq evalA(E ^ E', MT) = evalA(E, MT) xor evalA(E', MT) .
  eq evalA(E | E', MT) = evalA(E, MT) | evalA(E', MT) .
  var T : Token . var TS : TokenSequence .
  eq toCondExp(T '<< TS, MT) = toCondExp(T, MT) << toCondExp(TS, MT) .
endfm

fmod REXP-SEMANTICS is pr REXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
  eq evalB(E < E', MT) = (evalA(E, MT) < evalA(E', MT)) .
  eq evalB(E <= E', MT) = (evalA(E, MT) <= evalA(E', MT)) .
  eq evalB(E > E', MT) = (evalA(E, MT) > evalA(E', MT)) .
  eq evalB(E >= E', MT) = (evalA(E, MT) >= evalA(E', MT)) .
  eq evalB(E == E', MT) = (evalA(E, MT) == evalA(E', MT)) .
  eq evalB(E != E', MT) = (evalA(E, MT) =/= evalA(E', MT)) .
  var T : Token . vars TS1 TS2 : TokenSequence .
  eq toCondExp(T '< TS2, MT) = toCondExp(T, MT) < toCondExp(TS2, MT) .
endfm

fmod BEXP-SEMANTICS is pr BEXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable . var TS : TokenSequence .
  eq evalB(! E, MT) = not evalB(E, MT) .
  eq evalB(E && E', MT) = evalB(E, MT) and evalB(E', MT) .
  eq evalB(E || E', MT) = evalB(E, MT) or evalB(E', MT) .
  eq toCondExp('! TS, MT) = ! toCondExp(TS, MT) .
endfm

fmod CEXP-SEMANTICS is pr CEXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars C E E' : CondExp . var MT : MacroTable .
  ceq evalB(C ? E : E', MT) = evalB(E, MT) if evalB(C, MT) .
  ceq evalB(C ? E : E', MT) = evalB(E', MT) if not evalB(C, MT) .
endfm

fmod ALL-COND-EXP-SEMANTICS is
  pr DEF-COND-SEMANTICS .
  pr ARITH-EXP-SEMANTICS .
  pr BIT-EXP-SEMANTICS .
  pr REXP-SEMANTICS .
  pr BEXP-SEMANTICS .
  pr CEXP-SEMANTICS .
endfm

fmod SET-STRING is
  pr STRING .
  sort SetString .
  subsort String < SetString .
  op emptyS : -> SetString [ctor] .
  op __ : SetString SetString -> SetString [ctor assoc comm id: emptyS] .
  op _in_ : String SetString -> Bool .
  vars X X' : String . var S : SetString .
  eq X X = X .
  eq X in emptyS = false .
  eq X in (X' S) = X == X' or X in S .
endfm

fmod CFILES-MAP is
  pr SET-STRING . pr CPP-SYNTAX .
  sorts CFile CFilesMap CFilesMap? .
  subsort CFile < CFilesMap < CFilesMap? .
  op empty : -> CFilesMap [ctor] .
  op [_:_] : String LineSeq -> CFile [ctor] .
  op __ : CFilesMap? CFilesMap? -> CFilesMap? [ctor assoc comm id: empty] .
  op containsFileName : String CFilesMap? -> Bool .
```

```
  op dom : CFilesMap -> SetString .
  vars N N' : String .  var LS : LineSeq .
  vars FM FM1 FM2 : CFilesMap? . var FM' : CFilesMap .
  eq containsFileName(N, empty) = false .
  eq containsFileName(N, [N' : LS] FM) = (N == N') or containsFileName(N, FM) .
  eq dom(empty) = emptyS .
  eq dom([N : LS] FM') = N dom(FM') .
  op reach : CFilesMap? SetString -> SetString .
  op includedFiles : LineSeq -> SetString .
  var S : SetString .
  eq reach(empty, S) = emptyS .
  eq reach(FM, emptyS) = emptyS .
  ceq reach(FM, N) = emptyS  if not containsFileName(N, FM) .
  eq reach([N : LS] FM, N S)
     = includedFiles(LS) reach(FM, includedFiles(LS)) reach(FM, S) .
  var L : Line .
  eq includedFiles(nilLS) = emptyS .
  eq includedFiles((#include N cr) LS) = N includedFiles(LS) .
  eq includedFiles(L LS) = includedFiles(LS) [owise] .

  cmb ([N : LS] FM) : CFilesMap  if FM : CFilesMap /\ (not N in dom(FM)) /\
     (not N in (includedFiles(LS) reach(FM, includedFiles(LS)))) .
endfm

fmod CPP-STATE is
  pr MACRO-TABLE . pr TOKEN . pr CFILES-MAP .
  sorts CppState CppStateAttribute .
  subsort CppStateAttribute < CppState .
  op empty : -> CppState .
  op _,_ : CppState CppState  -> CppState [assoc comm id: empty] .
  op files : CFilesMap -> CppStateAttribute .
  op macroTbl : MacroTable -> CppStateAttribute .
  op curMacroCalls : IdentifierListP -> CppStateAttribute .
  op skip : Bool -> CppStateAttribute .
  op nestLevelOfSkipped : Nat -> CppStateAttribute .
  op branchTaken : Bool -> CppStateAttribute .
  op outputStream : TokenSequence -> CppStateAttribute .
endfm

fmod HELPING-OPS is
  pr CPP-STATE .
  op readFile : String CFilesMap -> LineSeq .
  vars F F' : String . var S : LineSeq . var FM : CFilesMap .
  eq readFile(F, empty) = nilLS .
  eq readFile(F, [F : S] FM) = S .
  ceq readFile(F, [F' : S] FM) = readFile(F, FM) if F =/= F' .
  op initialCppState : CFilesMap -> CppState .
  eq initialCppState(FM) = files(FM), macroTbl(empty),
        curMacroCalls(`(`)), skip(false), nestLevelOfSkipped(0),
        branchTaken(false), outputStream(nil) .
endfm

fmod LINE-SEQ-SEMANTICS is pr LINE-SEQ-SYNTAX .
  pr ALL-COND-EXP-SEMANTICS . pr CPP-STATE .
  op state : Line CppState -> CppState .
  op state : LineSeq CppState -> CppState .
  var L : Line .  var LS : LineSeq .  var S : CppState .
  var ILP : IdentifierListP . var I : Identifier . var IL : IdentifierList .
  var T : Token . vars TS AS O : TokenSequence . var MT : MacroTable .
  eq state(nil cr, S) = S .
  eq state(('## TS) cr, (curMacroCalls( (I, IL) ), skip(false), S))
    = state(TS cr, (curMacroCalls( (IL) ), skip(false), S)) .
  eq state(('## TS) cr, (curMacroCalls( (I) ), skip(false), S))
    = state(TS cr, (curMacroCalls( () ), skip(false), S)) .
  ceq state((T TS) cr, (macroTbl(MT), curMacroCalls(ILP), skip(false),
    outputStream(O), S))
    = state(TS cr, (macroTbl(MT), curMacroCalls(ILP), skip(false),
    outputStream(O T), S)) if not(isMacro(T, MT)) or (T in ILP) .
```

```
  ceq state((T '`( AS '`) TS) cr, (macroTbl(MT), curMacroCalls(ILP),
    skip(false), S))
    = state((expandWithArgs(MT[T], toTokenSeqList(AS)) '## TS) cr,
        (macroTbl(MT), curMacroCalls(cons(T, ILP)), skip(false), S))
    if isMacroWithArgs(T, MT) .
  ceq state((T TS) cr, (macroTbl(MT), curMacroCalls(ILP), skip(false), S))
    = state((expand(MT[T]) '## TS) cr,
        (macroTbl(MT), curMacroCalls(cons(T, ILP)), skip(false), S))
    if isMacroWithoutArgs(T, MT) .
  eq state((T TS) cr, (skip(true), S)) = skip(true), S .
  eq state(nilLS, S) = S .
  eq state(L LS, S) = state(LS, state(L, S)) .
endfm

fmod INCLUDE-SEMANTICS is pr INCLUDE-SYNTAX .
  ex LINE-SEQ-SEMANTICS . pr HELPING-OPS .
  var FN : String . var S : CppState .
  var FS : CFilesMap . var I : Identifier . var MT : MacroTable .
  eq state(#include FN cr, (files(FS), skip(false), S))
    = state(readFile(FN, FS), (files(FS), skip(false), S)) .
  eq state(#include FN cr, (skip(true), S))
    = skip(true), S .
  ceq state(#include I cr, (files(FS), macroTbl(MT), skip(false), S))
    = state(readFile(string(expand(MT[I])), FS),
    (files(FS), macroTbl(MT), skip(false), S)) if isMacroWithoutArgs(I, MT) .
  eq state(#include I cr, (skip(true), S))
    = skip(true), S .
endfm

fmod DEFINE-SEMANTICS is pr DEFINE-SYNTAX .
  ex LINE-SEQ-SEMANTICS .
  var I : Identifier . var TS : TokenSequence . var MT : MacroTable .
  var S : CppState . var IdL : IdentifierList . var MDD : MacroDefDir .
  eq state(#define I TS cr, (macroTbl(MT), skip(false), S))
    = macroTbl([I : (name I replText TS)] MT), skip(false), S .
  eq state(#define I ( IdL ) TS cr, (macroTbl(MT), skip(false), S))
    = macroTbl([I : (name I params (IdL) replText TS)] MT), skip(false), S .
  eq state(MDD, (skip(true), S)) = skip(true), S .
  ceq state(#undef I cr, (macroTbl(MT), skip(false), S))
    = macroTbl(remove(I, MT)), skip(false), S if isMacro(I, MT) .
  eq state(#undef I cr, (macroTbl(MT), skip(false), S))
    = macroTbl(MT), skip(false), S [owise] .
  eq state(#undef I cr, (skip(true), S)) = skip(true), S .
endfm

fmod COND-DIR-SEMANTICS is pr COND-DIR-SYNTAX .
  ex LINE-SEQ-SEMANTICS .
  pr ALL-COND-EXP-SEMANTICS .
  var N : Nat . var B : Bool . var AMT : MacroTable .
  var S : CppState . var I : Identifier . var TS : TokenSequence .

  --- Case 1 of #if: Not skipping -> Not skipping
  ceq state(#if TS cr, (macroTbl(AMT), skip(false), branchTaken(false), S))
    = macroTbl(AMT), skip(false), branchTaken(true), S
    if evalB(toCondExp(TS, AMT), AMT) = true .
  --- Case 2 of #if: Not skipping -> Skipping
  ceq state(#if TS cr, (macroTbl(AMT), skip(false), nestLevelOfSkipped(0),
        branchTaken(false), S))
    = macroTbl(AMT), skip(true), nestLevelOfSkipped(1), branchTaken(false), S
    if evalB(toCondExp(TS, AMT), AMT) = false .
  --- Case 3 of #if: Skipping -> Skipping
  eq state(#if TS cr, (skip(true), nestLevelOfSkipped(N), branchTaken(B), S))
    = skip(true), nestLevelOfSkipped(N + 1), branchTaken(false), S .

  --- Case 1 of #ifdef: Not skipping -> Not skipping
  ceq state(#ifdef I cr, (macroTbl(AMT), skip(false), branchTaken(false), S))
    = macroTbl(AMT), skip(false), branchTaken(true), S
    if isMacro(I, AMT) .
```

```
  --- Case 2 of #ifdef: Not skipping -> Skipping
  ceq state(#ifdef I cr, (macroTbl(AMT), skip(false), nestLevelOfSkipped(0),
        branchTaken(false), S))
    = macroTbl(AMT), skip(true), nestLevelOfSkipped(1), branchTaken(false), S
    if not isMacro(I, AMT) .
  --- Case 3 of #ifdef: Skipping -> Skipping
  eq state(#ifdef I cr, (skip(true), nestLevelOfSkipped(N), branchTaken(B), S))
    = skip(true), nestLevelOfSkipped(N + 1), branchTaken(false), S .

   --- Case 1 of #ifndef: Not skipping -> Not skipping
  ceq state(#ifndef I cr, (macroTbl(AMT), skip(false), branchTaken(false), S))
    = macroTbl(AMT), skip(false), branchTaken(true), S
    if not isMacro(I, AMT) .
  --- Case 2 of #ifndef: Not skipping -> Skipping
  ceq state(#ifndef I cr, (macroTbl(AMT), skip(false), nestLevelOfSkipped(0),
        branchTaken(false), S))
    = macroTbl(AMT), skip(true), nestLevelOfSkipped(1), branchTaken(false), S
    if isMacro(I, AMT) .
  --- Case 3 of #ifndef: Skipping -> Skipping
  eq state(#ifndef I cr, (skip(true), nestLevelOfSkipped(N), branchTaken(B), S))
    = skip(true), nestLevelOfSkipped(N + 1), branchTaken(false), S .

  --- Case 1 of #elif: Not skipping -> Skipping
  eq state(#elif TS cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S .
  --- Case 2 of #elif: Skipping -> Skipping
  ceq state(#elif TS cr, (macroTbl(AMT), skip(true), S))
    = macroTbl(AMT), skip(true), S  if evalB(toCondExp(TS, AMT), AMT) = false .
  --- Case 3 of #elif: Skipping -> Not skipping
  ceq state(#elif TS cr, (macroTbl(AMT), skip(true), nestLevelOfSkipped(1),
        branchTaken(false), S))
    = macroTbl(AMT), skip(false), nestLevelOfSkipped(0), branchTaken(true), S
        if evalB(toCondExp(TS, AMT), AMT) = true .

  --- Case 1 of #else: Not skipping -> Skipping
  eq state(#else`cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S .
  --- Case 2 of #else: Skipping -> Skipping
  eq state(#else`cr, (skip(true), nestLevelOfSkipped(N), branchTaken(true), S))
    = skip(true), nestLevelOfSkipped(N), branchTaken(true), S .
  --- Case 3 of #else: Skipping -> Not skipping
  eq state(#else`cr, (skip(true), nestLevelOfSkipped(1), branchTaken(false), S))
    = skip(false), nestLevelOfSkipped(0), branchTaken(true), S .

  --- Case 1 of #endif: Not skipping -> Not skipping
  eq state(#endif`cr, (skip(false), branchTaken(true), S))
    = skip(false), branchTaken(false), S .
  --- Case 2 of #endif: Skipping -> Skipping
  ceq state(#endif`cr, (skip(true), nestLevelOfSkipped(N), S))
    = skip(true), nestLevelOfSkipped(N - 1), S   if N > 1 .
  --- Case 3 of #endif: Skipping -> Not Skipping
  eq state(#endif`cr, (skip(true), nestLevelOfSkipped(1), branchTaken(true), S))
    = skip(false), nestLevelOfSkipped(0), branchTaken(false), S .
endfm

fmod CPP-SEMANTICS is
  pr CPP-SYNTAX . pr HELPING-OPS . pr LINE-SEQ-SEMANTICS .
  pr INCLUDE-SEMANTICS . pr DEFINE-SEMANTICS . pr COND-DIR-SEMANTICS .
  op preprocess : CFilesMap String -> TokenSequence .
  op returnOutput : CppState -> TokenSequence .

  var F : String . var LS : LineSeq . var FM : CFilesMap .
  var O : TokenSequence . var S : CppState .
  eq preprocess([F : LS] FM, F)
    = returnOutput(state(LS, initialCppState([F : LS] FM))) .
  eq preprocess(FM, F) = nil [owise] .
  eq returnOutput(outputStream(O), S) = O .
endfm
```